

1 Setting

1.1 vimrc

```
1 set ts=4 sts=4 sw=4
2 set ai si nu
```

2 Math

2.1 basic arithmetic

```
1 typedef long long ll;
2 typedef unsigned long long ull;
3
4 // calculate ceil(a/b)
5 // |a|, |b| <= (2^63)-1 (does not cover -2^63)
6 ll ceildiv(ll a, ll b) {
7     if (b < 0) return ceildiv(-a, -b);
8     if (a < 0) return (-a) / b;
9     return ((ull)a + (ull)b - 1ull) / b;
10 }
11
12 // calculate floor(a/b)
13 // |a|, |b| <= (2^63)-1 (does not cover -2^63)
14 ll floordiv(ll a, ll b) {
15     if (b < 0) return floordiv(-a, -b);
16     if (a >= 0) return a / b;
17     return -(ll)((ull)(-a) + b - 1) / b;
18 }
19
20 // calculate n^k % m
21 ll modpow(ll n, ll k, ll m) {
22     ll ret = 1;
23     n %= m;
24     while (k) {
25         if (k & 1) ret = ret * n % m;
26         n = n * n % m;
27         k /= 2;
28     }
29     return ret;
30 }
31
32 // calculate gcd(a, b)
33 ll gcd(ll a, ll b) {
34     return b == 0 ? a : gcd(b, a % b);
35 }
36
37 // find a pair (c, d) s.t. ac + bd = gcd(a, b)
38 pair<ll, ll> extended_gcd(ll a, ll b) {
39     if (b == 0) return { 1, 0 };
40     auto t = extended_gcd(b, a % b);
41     return { t.second, t.first - t.second * (a / b) };
42 }
43
```

```
44 // find x in [0,m) s.t. ax === gcd(a, m) (mod m)
45 ll modinverse(ll a, ll m) {
46     return (extended_gcd(a, m).first % m + m) % m;
47 }
48
49 // calculate modular inverse for 1 ~ n
50 void calc_range_modinv(int n, int mod, int ret[]) {
51     ret[1] = 1;
52     for (int i = 2; i <= n; ++i)
53         ret[i] = (ll)(mod - mod/i) * ret[mod%i] % mod;
54 }
```

2.2 sieve methods : prime, divisor, euler phi

```
1 // find prime numbers in 1 ~ n
2 // ret[x] = false -> x is prime
3 // O(n*loglogn)
4 void sieve(int n, bool ret[]) {
5     for (int i = 2; i * i <= n; ++i)
6         if (!ret[i])
7             for (int j = i * i; j <= n; j += i)
8                 ret[j] = true;
9 }
10
11 // calculate number of divisors for 1 ~ n
12 // when you need to calculate sum, change += 1 to += i
13 // O(n*loglogn)
14 void num_of_divisors(int n, int ret[]) {
15     for (int i = 1; i <= n; ++i)
16         for (int j = i; j <= n; j += i)
17             ret[j] += 1;
18 }
19
20 // calculate euler totient function for 1 ~ n
21 // phi(n) = number of x s.t. 0 < x < n && gcd(n, x) = 1
22 // O(n*loglogn)
23 void euler_phi(int n, int ret[]) {
24     for (int i = 1; i <= n; ++i) ret[i] = i;
25     for (int i = 2; i <= n; ++i)
26         if (ret[i] == i)
27             for (int j = i; j <= n; j += i)
28                 ret[j] -= ret[j] / i;
29 }
```

2.3 primality test

```
1 bool test_witness(ull a, ull n, ull s) {
2     if (a >= n) a %= n;
3     if (a <= 1) return true;
4     ull d = n >> s;
5     ull x = modpow(a, d, n);
6     if (x == 1 || x == n-1) return true;
7     while (s-- > 1) {
8         //x = large_mod_mul(x, x, n);      TODO!!
9         x = x * x % n;
```

```

10         if (x == 1) return false;
11         if (x == n-1) return true;
12     }
13     return false;
14 }
15
16 // test whether n is prime
17 // based on miller-rabin test
18 // O(logn*logn)
19 bool is_prime(ull n) {
20     if (n == 2) return true;
21     if (n < 2 || n % 2 == 0) return false;
22
23     ull d = n >> 1, s = 1;
24     for(; (d&1) == 0; s++) d >>= 1;
25
26 #define T(a) test_witness(a##ull, n, s)
27     if (n < 4759123141ull) return T(2) && T(7) && T(61);
28     return T(2) && T(325) && T(9375) && T(28178)
29         && T(450775) && T(9780504) && T(1795265022);
30 #undef T
31 }

```

2.4 chinese remainder theorem

```

1 // find x s.t. x === a[0] (mod n[0])
2 //             === a[1] (mod n[1])
3 //             ...
4 // assumption: gcd(n[i], n[j]) = 1
5 ll chinese_remainder(ll* a, ll* n, int size) {
6     if (size == 1) return *a;
7     ll tmp = modinverse(n[0], n[1]);
8     ll tmp2 = (tmp * (a[1] - a[0]) % n[1] + n[1]) % n[1];
9     ll ora = a[1];
10    ll tgcd = gcd(n[0], n[1]);
11    a[1] = a[0] + n[0] / tgcd * tmp2;
12    n[1] *= n[0] / tgcd;
13    ll ret = chinese_remainder(a + 1, n + 1, size - 1);
14    n[1] /= n[0] / tgcd;
15    a[1] = ora;
16    return ret;
17 }

```

2.5 burnside's lemma

경우의 수를 세는데, 특정 transform operation(회전, 반사, ...)해서 같은 경우들은 하나로 친다. 전체 경우의 수는?

- 각 operation마다 이 operation을 했을 때 변하지 않는 경우의 수를 센다 (단, “아무것도 하지 않는다”라는 operation도 있어야 함!)

- 전체 경우의 수를 더한 후, operation의 수로 나눈다. (답이 맞다면 항상 나누어 떨어져야 한다)

3 Data Structure

3.1 fenwick tree